1. Type System for Prolog

Sucess criteria: implement a basic modification of the Hindley-Milner type system adapted to logic programming langauges. This is described in *A polymorphic type system for Prolog, Alan Mycroft, Richard A. O'Keefe*: `https://www.sciencedirect.com/science/article/pii/0004370284900171`.

After completing this I could extend the project by:

- Extending the type checker to support higher order programming, in the form of a call N predicate
- Extending the type system to include type classes

These extensions would mainly follow the thesis *Expressive Type Systems for Logic Programming Languages*: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.2178`

Pros:

- Can be pretty confident of implementing sucess criteria
- It's cool

Cons:

- Difficult to evaluate quantitively, so I risk having a weak evaluation section
- Extensions mainly follow one PhD thesis, there may be flaws in this thesis

2. Prolog Compiler

Sucess criteria: implement a lexer and parser for prolog that outputs VM code. Implement a VM to interpret this output VM code.

This could be extended by attempting some optimisations from project idea 3.

Pros:

- Can evaluate performance of VM to get graphs for evaluation section
- The project seems the safest of the 3 options
- By putting optimisations as extension I have the potential to do something interesting, but the risk is smaller

Cons:

- My implementation will not do anything special compared to standard Prolog implementations, it will just be worse
- The interesting work is in the extension, so the project is a bit boring if I don't get to the extension

3. Comparison of Optimisations to Prolog Programs

Sucess criteria: implmenent a Prolog interpreter, and compare the effect of at least two optimisations for this interpreter.

Optimisations I could perform include:

- Use determinancy annotations or inference to prevent backtracking through determinate goals. I could probably verify these annotations.
- Use mode annotation or inference to speed up unification. I could also work on checking the mode annotations.
- Use sucess annorations (not inference) to indicate when a goal always succeeds. This means you don't need to return a bool to indicate success. I'm not sure how you could verify these annotations.
- Detect shared structures to skip occurs check if there is no sharing
- Compare different search techniques (heuristics?)
- Experimentation with basic parallelism
- Tail call optimisation when all predicates are determinate except the last
- If I use interpreted byte code then I could try simple push/pop pair elimination etc (but then I need to have interpreted byte code)
- Use a type checker to remove run-time type checks? i.e. to speed up unification since you can't unify things of different types (so without type checking you'd have to do a run-time type check for unification)

Most of these are taken from `https://core.ac.uk/download/pdf/82661515.pdf`.

Pros:

- Sucess criteria is pretty vague so I shouldn't fail :D
- More interesting than option 2
- Comparison of optimisations lends itself easily to quantative evaluation
- I have a lot of options for optimisation, and there are probably lots more out there

Cons:

- I don't really understand most of the optimisations I have suggested
- I may not see performance gains from the optimisations, making the write up difficult
- Right now I won't be able to write a good time plan for this, since I don't know how long each optimisation would take me
- I don't want to commit to any specific optimisations in case they don't work out, so I don't know what to focus on
- I will need to write my own Prolog back end (I can't find one to use that I trust), which could end up taking a big chunk of time
- In order to compare many optimisations I would want to keep the subset of Prolog I am using small. This may limit the improvement I get (eg determinancy analysis works well in presence of cut, but cut is impure)
- Unlikely to get round to implementing a type system :'(